Distributed Sequential Computing Using Mobile Code: Moving Computation to Data

Lei Pan, Lubomir F. Bic, and Michael B. Dillencourt
Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
{*pan,bic,dillenco*}*@ics.uci.edu*

Abstract

Sequential computations can benefit from a distributed environment consisting of a network of workstations through the use of a mobile agent system. We found significant performance improvement when sequential algorithms for solving large industrial problems are implemented using mobile agents. This is because raw data is distributed so that the cost of disk paging is completely eliminated, and a principle of "code moving to data" is followed to achieve efficient communication through the network. We argue that mobile agent systems provide a new level of abstraction in which application programming is made easier because the sequential algorithms remain essentially unchanged in mobile agent code.

Keywords: *autonomous mobile agents, MESSENGERS, Network of Workstations, distributed computing, distributed sequential computing, matrix operations, numerical solution of linear system of equations, Gauss-Seidel iteration, Crout factorization*

1. Introduction

One of the major foci of distributed computing is using networks of workstations to solve numerical problems efficiently. The most obvious way of achieving this is through parallel computation, when the problem lends itself to parallelization. This can make the computation efficient in several ways. Speedup is achieved by distributing the total computational work over multiple workstations; this is the fundamental advantage of parallelism. But in addition, the computation on each workstation may use a smaller working set, leading to fewer page faults. Since processing page faults is part of the work that must be performed, distributing a numerical computation over multiple workstations thus decreases the total amount of work required by the computation. In some cases, the efficiency advantages due to the second effect—reducing total work—may dwarf the efficiency advantages due to distributing the workload. This suggests that even for algorithms that are not easily parallelized, distributing the data over multiple workstations and running the sequential algorithm can potentially result in significant increases in efficiency. Of course, it is often possible to obtain further improvement with a parallel re-implementation. The advantage of distributing the sequential algorithm is that it can lead to a significant improvement in situations where paging is occurring with only a very minor re-programming effort; the question of whether the additional improvement to be gained from a parallel re-implementation justifies what might be a major re-programming effort would then need to be evaluated on a case-by-case basis.

In this paper, we exploit this observation to provide efficient implementations for solving some numerical problems. Our algorithms run on a network of workstations. We call our approach *distributed sequential computing*. The algorithms are sequential in the sense that there is always a single locus of computation. The data is distributed; in essence, we use the network as a data farm. Rather than having the data migrate to the code (the classical message-passing approach), the code follows the data (an agent-based approach).

Our approach has two major advantages:

Improved Performance: If the size of the state of the program is small relative to the amount of data, it is cheaper to move the code to the data (i.e., to send the program and the associated state information) than to move the data to the code. Moreover, the agent doing the computation can send out additional agents to other machines to pre-fetch data needed for subsequent computation and to post-write data that has already been computed. In this way, the program can avoid paging overhead even when the amount of data that needs to be processed is much larger than the amount of memory available in the entire network.

Algorithmic Integrity: An existing sequential algorithm does not have to be substantially rewritten; it merely

needs to be augmented with the commands to hop with an agent from one machine to another at the appropriate points in the algorithm.

The agent-based nature of the approach described in this paper distinguishes it from previous work aimed at defeating disk paging by using remote memory on networked machines. A description of this approach is presented by Dramitinos and Markatos in [4]. This is a very promising approach because future improvements in the performance of disk paging appear to be limited by the inherent bottleneck of mechanical seeking time, but there seems to be no limit on the speed and bandwidth of computer network. Dramitinos and Markatos have implemented an "operating system" which does swapping using remote memory ([4]), and they found it to be "up to twice as fast as traditional disk paging." However, the limiting factor in their approach is the amount of data that needs to be moved: it is faster to move a small amount of code and data to a huge amount of data than to perform the converse move.

In this paper, we describe agent-based implementations of three classical sequential numerical algorithms—matrix multiplication and two different algorithms for solving linear system of equations. The reasons for us to choose sequential algorithms are as follows: 1). As will be clear later in this paper, on large problems sequential algorithms benefit greatly from good use of a distributed environment; 2). There are algorithms that can not be easily parallelized, and yet are practically useful, and existing software needs to take full advantage of a distributed environment with little porting effort. Our implementations use MESSENGERS, a system for agent-based distributed computing developed in Information and Computer Science at the University of California, Irvine ([1, 5]). All the performance tests are run on SUN Ultra Sparc 1 model 170's with 64MB of main memory, 1GB of virtual memory, and 10Mbps of Ethernet connection. These workstations have a shared file system.

The remainder of this paper is organized as follows. Section 2 contains a brief overview of an agent-based system: MESSENGERS. The agent-based implementation of matrix multiplication is presented in Section 3, while Section 4 discusses the implementation of two methods for solving linear systems: Gauss-Seidel iterative method and Crout factorization based direct method.

2. The MESSENGERS mobile agent system

Mobile agents are programs that move dynamically among networked machines, carrying their data and execution states with them. A mobile agent, with strong mobility, can halt its execution, encapsulate the values of its variables and execution stack, move to another machine, restore the state, and continue executing. Although ultimately based on message passing at low level, this "mobile" capability, as will be shown later in this paper, is the right vehicle for application programmers who want to make good use of a distributed environment with less efforts.

All mobile-agent systems have the same general architecture: a server on each machine accepts incoming agents, and for each agent, starts an appropriate execution environment, loads the agent's state information into the environment, and resumes agent execution. The most important feature that distinguishes agent-based systems from other conventional systems is that all functionality of the application is embedded in individual agents, i.e., the programs are carried by agents as they navigate through space.

The MESSENGERS system ([5]) is an environment for distributed computing in which applications are developed as collections of autonomous self-migrating computations, called Messengers. In the MESSENGERS system, there are three levels of networks. The lowest level is the physical network(e.g. a LAN or WAN), which constitutes the underlying computational nodes. Superimposed on the physical layer is the daemon network, where each daemon is a server process that receives, executes, and dispatches Messengers. The logical network is an application-specific computation network created on top of the daemon network. Messengers may be injected (from the shell or by another Messenger) into any of the daemon nodes and they may start creating new logical nodes and links on the current or any other daemons.

The important concepts and features of MESSENGERS are as follows:

(1) All participating physical nodes have MESSENGERS daemons running on them. The logical network is thus established on top of this daemon network, which in turn runs on the physical network.

(2) MESSENGERS scripting language is a subset of C, so it is easy for C/C++ programmers to develop MESSENGERS applications. Furthermore, the MESSENGERS script is first preprocessed into C code which is then compiled into machine native code, and therefore the execution is very efficient.

(3) There are two types of variables: Messenger variables and node variables. A Messenger variable, often taken as a medium for communication, is one that belongs to a particular Messenger and travels with that Messenger to different logical nodes, whereas a node variable is one that is stationary to a logical node.

(4) The most important navigational statement for Messengers are $create()$ and $hop()$. The $create()$ statement generates a link along which a Messenger moves, and creates a logical node on the physical node. The $hop()$ statement causes the Messenger to navigate to a node, along a link or using the node's logical address.

```
(1) for (i=1; i<=N; i++) {
(2)   for (j=1; j<=N; j++) {
(3)     C[i][j] = 0.0;
(4)     for (k=1; k<=N; k++) {
(5)       C[i][j] += A[i][k] * B[k][j];
(6)     }
(7)   }
(8) }
```

Figure 1. Pseudocode for sequential matrix multiplication

3. Sequential matrix multiplication implemented using MESSENGERS

The primary advantage of distributed computing on networks of workstations is the availability of inexpensive and yet abundant resources such as multiple CPU's, collective memory and disk space, and network connections. A major challenge is building tools to help develop applications that take full advantage of these resources. The tools have to be easy to use for those application programmers, such as software engineers in CAE business or researchers in numerical analysis, who may know little about details of the distributed environment. The tools have to also ensure that significant changes are avoided when porting existing code or algorithm to a distributed environment. In this section, we provide a very simple example, namely sequential matrix multiplication, to demonstrate why mobile agent systems such as MESSENGERS are among these useful tools.

Sequential pseudocode for the multiplication of two square dense matrices is listed in Figure 1.

When the total memory size of the three matrices exceeds that of the main memory on one workstation, virtual memory is used and therefore disk paging occurs. Disk paging can dramatically degrade the performance. The exact point at which the performance deteriorates may vary depending on the details of the implementation and the data access patterns, but the performance falloff is quite severe once heavy paging kicks in. Figure 5 shows a sudden jump of the elapsed time with the dashed line. To eliminate disk paging, we use two connected machines, and program the above algorithm in MESSENGERS. We first decompose the two matrices into horizontal and vertical strips, as shown in Figure 2.

With this, each piece of C_{ij} will be the multiplication of two sub-matrices A_i and B_j. The pseudocode in Figure 1 can be slightly modified to work in this block fashion.

Now we construct a logical network on two workstations

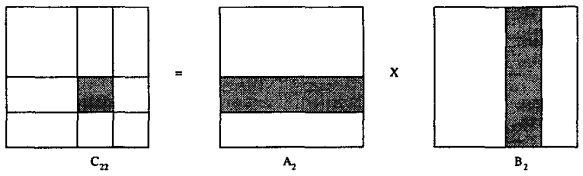

Figure 2. Matrix decomposition

as shown in Figure 3. The logical network is linked bidirectionally so that a Messenger can hop from one node to the other if it so decides.

Figure 3. Logical network used for matrix multiplication

We have three Messengers to carry out the task. One Messenger called Multiplier will do the actual multiplication, calculating C_{ij} on the logical node where A_i and B_j reside. So it is easy to imagine that the Messenger Multiplier hops to the two nodes alternately, carrying its code, and computation state (which indicates which C_{ij} is being calculated). The computation locus is only on a single node, either Node 1 or Node 2, at any particular time. Another Messenger, called Preloader, injected by Multiplier on one node, hops to the other and preloads the node with proper A_i and B_j. A third Messenger named Postwriter writes out the computed submatrix C_{ij} to the hard disk, after the Multiplier finishes the computation of C_{ij} and hops away. These three Messengers work together to pipeline all the disk I/O (reading from or writing to hard disk typically takes much less time than multiplying two submatrices), and to minimize communication. Only code, which is tiny compared to the data, migrates between the two nodes. It is easy to see that provided the decomposition is done right (in other words, provided the total size of A_i, B_j and C_{ij} does not exceed the size of the main memory on either node), the submatrix computation will never cause disk paging.

Figure 4 is the pseudocode in MESSENGERS ($matrix\_mult()$ is a function that does the multiplication in the block fashion).

As can be seen clearly from Figure 4, the MESSENGERS implementation leaves the sequential algorithm essentially unchanged: the only thing added is the $hop()$ statement and the injections of a couple of other Messengers (through the $preload()$ and $postwrite()$ functions) which simply add few lines of code.

```
(1) preload(this, A(1));
(2) preload(this, B(1));
(3) for (i=1; i<=num_pieces; i++) {
(4)   preload(other, A(i));
(5)   for (j=1; j<=num_pieces; j++) {
(6)     if (j < num_pieces)
(7)       preload(other, B(j+1));
(8)     else {
(9)       if (i < num_pieces) {
(10)        preload(other, A(i+1));
(11)        preload(other, B(1));
(12)      }
(13)    }
(14)    C(i,j) = matrix_mult(A(i), B(j));
(15)    hop(other);
(16)    postwrite(other, C(i,j));
(17)  }
(18) }
```

Figure 4. Pseudocode for matrix multiplication in MESSENGERS

Figure 5 shows that the performance of the sequential C implementation deteriorates dramatically after the total size of the matrices reaches a certain critical value. Our MESSENGERS implementation has performance almost identical to that of the C implementation when the total size of the matrices is below this critical value, and it continues the same performance trend after the total size exceeds this critical value.

4. Solving large linear system of equations sequentially in a distributed environment using MESSENGERS

Much of the CPU time spent in running numerical analysis is used in solving linear systems of equations. In this section we consider two classical methods for solving linear systems: namely, an iterative method based on Gauss-Seidel iteration and a direct method based on Crout factorization. We describe our MESSENGERS implementations of these methods that use multiple machines to eliminate paging overhead, while leaving the original sequential algorithms essentially unchanged.

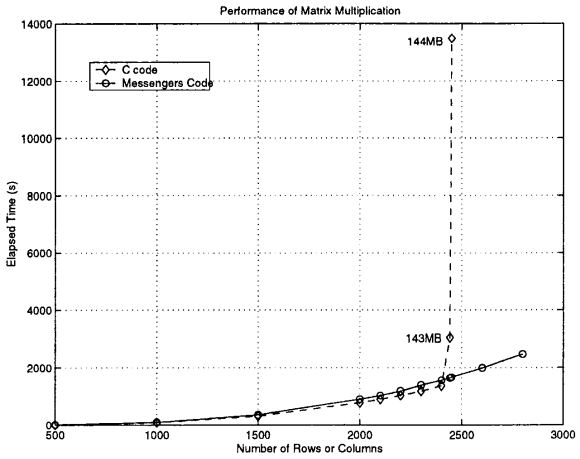

Figure 5. Performance of matrix multiplication

4.1. Iterative method based on Gauss-Seidel iteration

Let $K\underline{u} = \underline{f}$ be a system of linear equations, where K is a $N \times N$ matrix, $\underline{u}$ and $\underline{f}$ are vectors of size N. Matrix K can be decomposed into $K = D - L - U$, where D is the diagonal of K, and $-L$ and $-U$ are the strictly lower and upper triangular parts of K.

If we define the Gauss-Seidel iterative matrix P_G by $P_G = (D - L)^{-1}U$, we can express Gauss-Seidel iterative method as ([2])

$$\underline{u}_{n+1} \leftarrow P_G\underline{u}_n + (D - L)^{-1}\underline{f}. \quad (1)$$

We update the components of $\underline{u}$ in ascending order. Components of the new approximation are used as soon as they are computed. In other words, we solve the j^{th} equation for u_j using new approximations for components $1, 2, ..., j-1$. To start the iteration, we use an arbitrary vector as our initial guess, e.g. $\underline{u}_0 = [0, 0, ...0]^T$. Notice that Gauss-Seidel iteration can be done in a block fashion, in which only a portion of the solution vector is updated given a slice of the matrix K and the entire solution vector from the previous iteration. This is illustrated in Figure 6.

Since matrix K is usually sparse and banded, we only store its nonzero terms. These terms will be stored in a 1D array, in the compact row storage scheme. An integer array of size N is used to store the positions, in the 1D array, of the first nonzero terms of all rows.

The matrix K is decomposed into horizontal slices, as illustrated in Figure 6. Each slice of K is used to update the components of $\underline{u}$ corresponding to row positions of the slice. The logical network used by the MESSENGERS implementation is a ring, where the number of nodes is equal

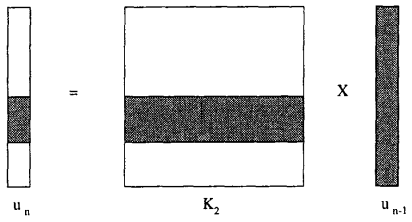

Figure 6. Matrix decomposition and block fashion of Gauss-Seidel iteration

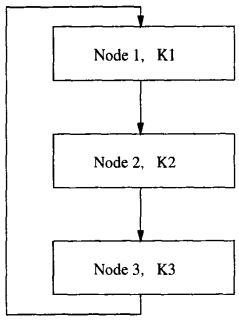

Figure 7. The logical network for Gauss-Seidel iterative method

```
(1) preload(all nodes);
(2) j=1;
(3) stop = 0;
(4) while (!stop) {
(5)   status = block_gs(K(j), u, f);
(6)   if (j == LAST_PIECE) {
(7)     err = check_error(u);
(8)     if (err < TOL)
(9)       stop = 1;
(10)    else
(11)      j = 0;
(12)  }
(13)  j ++;
(14)  hop(next);
(15) }
```

Figure 8. Pseudocode for MESSENGERS implementation of Gauss-Seidel iteration

to the number of slices into which K is decomposed; the case of 3 nodes is shown in Figure 7. A Messenger carrying the solution vector can visit a node, use the slice of K on that node to update the corresponding solution components (this corresponds to the block fashion mentioned earlier), and then hop to the next node, repeating this until the error satisfies the stopping criteria. The pseudocode is shown in Figure 8. The function $block\_gs()$ runs one step of Gauss-Seidel iteration (1) in the block fashion. Notice that the MESSENGERS code is almost identical to the sequential code that would be written once the matrix K and the vector $\underline{u}$ are partitioned into strips as described above; the only difference is the hop statement.

Performance data is shown in Figure 9. The K matrices used are sparse and banded, and their bandwidth is 10% of their dimensions. The numbers in parentheses by the MESSENGERS curve indicate the number of workstations used, and the amount of total memory required is marked by the C code curve. It would also be possible to implement the MESSENGERS version of Gauss-Seidel method using only two workstations, with one pre-fetching while the other performs the computation, similar to the implementation of matrix multiplication described in the previous section. We chose the implementation shown here to demonstrate the concept of having the code and state following raw data that is distributed but stationary.

4.2. Direct method based on Crout factorization

In the system $K\underline{u} = \underline{f}$, when matrix K is symmetric and positive-definite, there exists a non-singular lower triangular matrix L, with unit diagonal entries, and diagonal matrix D such that $K = LDL^T$. The process of obtaining matrices L and D is called *factorization*. After factorization, the solution is carried out in three steps, namely Forward Reduction, Diagonal Scaling, and Backward Substitution.

One possible way of conducting factorization is due to Crout [7], and the pseudocode is listed in Figure 10. Only the upper part of the matrix K is stored due to its symmetry. To further save storage for a sparse and banded matrix, a "skyline" storage scheme is used, in which we link the first non-zero items in every column together to form a "skyline," and we do not store the zero values above this line. Zero values under the skyline are stored, since they could become non-zero during Crout factorization. The columns under the skyline are stored in a single 1D double precision array, and another 1D integer array of size N is used to store the locations of the diagonal terms in the big double precision array. Note that no additional storage is necessary in the factorization process since K is overwritten by U and D.

In line (3) of Figure 10, the summation over l corresponds to a dot product of two sub-vectors of columns i and j. These are the two shaded vectors in Figure 11. The

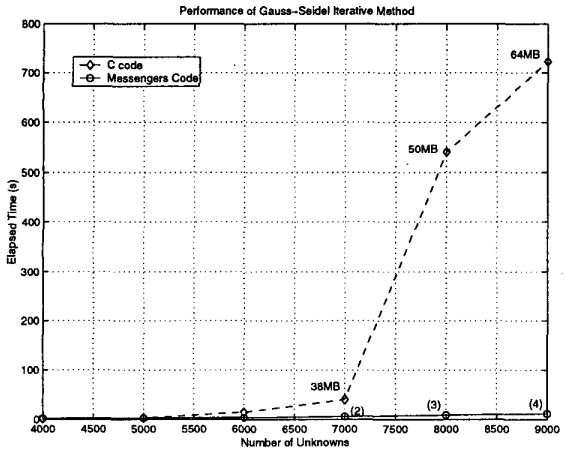

Figure 9. Performance of Gauss-Seidel iterative method

```
(1)  For j = 1...N
(2)      For i = 2...j-1
(3)          K_ij ← K_ij − Σ_{l=1}^{i-1} K_li K_lj
(4)      End For
(5)      For i = 1...j-1
(6)          T ← K_ij
(7)          K_ij ← T / K_ii
(8)          K_jj ← K_jj − T K_ij
(9)      End For
(10) End For
```

Figure 10. Pseudocode for Crout factorization

computation of the j^{th} column would depend on the previously computed columns, called the "working set." Figure 12(a) is an example for a banded matrix. The shaded area is the working set for the j^{th} column.

Crout factorization is an algorithm with very good locality of access. There does not need to be sufficient memory to hold the entire half matrix; as long as the working set fits in memory, performance is good. However, when the size of the working set exceeds the size of the main memory on a single workstation, extensive paging overhead occurs. In Figure 15 the dashed line shows how bad the performance can be when the size of the working set exceeds the size of main memory.

There are two ways to handle the access to the working set in a distributed implementation. One way is to bring the

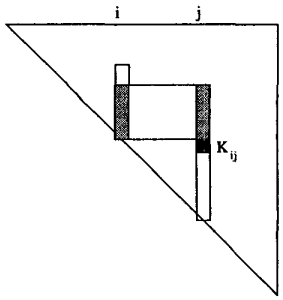

Figure 11. Computing of K_{ij} requires the dot product of the two shaded vectors

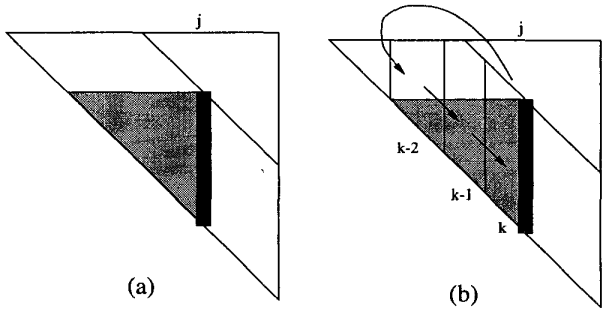

Figure 12. (a) Working set for the j^{th} column (b) Working set decomposition

columns from remote memory (or the hard disk, in the presence of paging) to the main memory where the j^{th} column is computed, as they are needed. This will result in significant communication overhead because the entire working set is moved.

The other way is the agent-based approach. The basic idea is to move the code and the j^{th} column to the working set, which is distributed among several workstations, and compute the terms of the column on remote machines. The amount of communication overhead incurred by this approach is much smaller than with the first approach, because a single column is much smaller than the entire working set.

The working set is decomposed into several pieces, where the number of pieces is chosen so that each piece can fit into the main memory of one workstation. Figure 12(b) shows an example for which the working set is subdivided into three pieces. The arrows indicate how an agent, carrying the j^{th} column which it is computing, would move through the pieces of the working set.

Figure 13(a) shows the logical network, again assuming that the working set is decomposed into three pieces. The logical network consists of four nodes. Three of the nodes

are used to hold the three working set pieces shown in Figure 12(b), and a fourth node is used to pre-fetch the next piece that will be computed later. All four logical nodes are fully connected, so that an agent can hop to any node from anywhere in one step. As indicated by the arrows, an agent would carry the j^{th} column in its Messenger variable, and hop to logical node 3 where piece $k-2$ resides; after it finishes computing using piece $k-2$, it hops forward along the link to node 2 where piece $k-1$ resides (note that $k-2$ and $k-1$ are pieces that have been already computed in the previous loops); finally, it hops back to node 1, computes the rest of column j using piece k, and then unloads the j^{th} column from its Messenger variable to the node variable. These loops go on and on until all columns in piece k are computed, at which time the agent moves on to the next node (4 in the example) to compute piece $k+1$. Again, the previously computed pieces $k-1$ and k will be used in computing $k+1$, but piece $k-2$ will no longer be used in factorization, so we can now save piece $k-2$ to the hard disk, and use node 3 to pre-fetch piece $k+2$. Figure 13(b) shows this next step when piece $k+1$ is being computed. If we compare Figure 13(b) with Figure 13(a), we see that the logical network ring is like a "running wheel" rotating forward (clockwise) while it processes the matrix pieces in sequence. In order for an agent to hop to the right machine for a column, a mapping from a column number to a piece number to a logical node number is kept on every node.

The Crout algorithm in Figure 10 is augmented into the MESSENGERS implementation shown in Figure 14. Three hops and three load/unloads are added. Note that the cost of the hop statement will be negligible if the destination node happens to be the one that the Messenger resides on. An injection of pre- and post-fetching Messengers is also added. We gain a huge advantage by doing this, since now our program can solve problems that are many times (35 in our example) larger than the memory available on a single workstation with only few workstations (4 in our example). We would have had to use 35 machines if we had not used pre-fetching.

Figure 15 shows the performance of Crout factorization. The K matrices used are sparse and banded, and their bandwidth is 10% of their dimensions. The numbers in parentheses by the MESSENGERS curve indicate the number of pieces into which the matrix K is subdivided. Total memory required is also marked for some problem sizes. One may notice that the gap between the two curves (excluding the segment corresponding to disk paging) now is larger than what's shown in Figure 5 (matrix multiplication example). This is because the amount of communication through the network is much larger. In fact, since almost all the columns (except for the columns in the very first piece) are carried around by the Messenger, the amount of communication is proportional to the size of the entire matrix

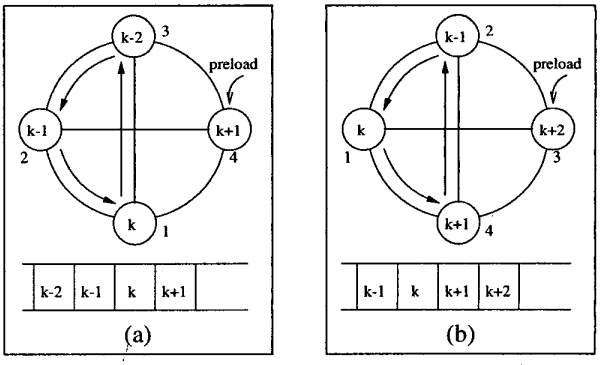

Figure 13. Logical network for Crout factorization. (a) currently computing piece k. (b) currently computing piece $k+1$.

($O(N^2)$). However, since the computation complexity of Crout factorization method is $O(N^3)$, our implementation is scalable. Also, with fast network (100Mbps), our analysis and test show that the time for communication overhead can be reduced to about 5%, as compared to the current 30%, of the total elapsed time.

5. Conclusions and final remarks

We have presented three examples of mobile-agent-based distributed numerical computations on a network of workstations. In each case, the performance on a very large problem, using considerably more data than can fit in the memory of a single machine, was quite close to the performance that would have been achieved if the data fit in memory. These performance tests are done on a relatively slow network: 10 Mbps Ethernet. In addition, the MESSENGERS implementations preserved the algorithmic integrity: the algorithms remain essentially unchanged—the main difference between the MESSENGERS implementation and the sequential algorithm is the addition of hop statements to allow the MESSENGERS programs to become mobile in the distributed environment.

Producing mobile-agent based implementations such as the ones described in this paper involves analyzing the data access pattern; decomposing the data, distributing the pieces, and managing the data-node mapping; and arranging the order of data access in accordance with the order of computing. All of these steps, or something quite similar, are part of other distributed programming paradigms such as message passing. The resulting code changes in mobile code are very few, because the mobile agent system has helped us at a higher level that is close to the application level. It is worth emphasizing that with sequential algorithms, such as the ones presented here, we do not have

```
(1)     For j = 1...N
(1.1)     hop(to column j)
(1.2)     load column j
(2)       For i = 2...j-1
(2.1)       hop(to column i)
(3)         K_ij ← K_ij - Σ_{l=1}^{i-1} K_li K_lj
(3.1)       load K_ii
(4)       End For
(4.1)     hop(to column j)
(5)       For i = 1...j-1
(6)         T ← K_ij
(7)         K_ij ← T / K_ii
(8)         K_jj ← K_jj - T K_ij
(9)       End For
(9.1)     unload column j
(9.2)     inject I/O Messenger if required
(10)    End For
```

Figure 14. Pseudocode for MESSENGERS implementation of Crout factorization

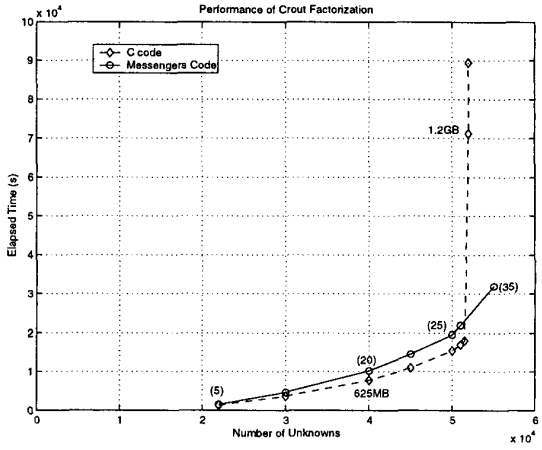

Figure 15. Performance of Crout factorization

to look for any computation parallelization, and yet we still gain huge speedup for large problems in a distributed environment. Of course, all the algorithms we tried in this paper have their parallel equivalences, with some being easy to implement (e.g. the iterative method) and others (e.g. Crout factorization) requiring significant amount of re-work from their sequential counterparts ([3]).

Using a network of workstations as a data paging farm is a promising notion. The mobile-agent-based approach borrows this idea, but it distinguishes itself from the one presented in [4] in two ways. First, we do not move all data to one single machine; rather we move computation to data, which can significantly reduce communication overhead. It is this moving of sometimes huge amounts of data that limits the paging device presented in [4] to work well only with high bandwidth networks. Second, we utilize the "spare" CPU cycles in the workstation farm to parallelize data I/O, which in our examples hides the cost of disk I/O completely. This enables us to use only a few workstations to solve very large problems, with total data size much larger than the total amount of main memory available in the workstation farm.

The agent-based approach described here is at a higher level than the paging farm approach, which is at the operating-system level: the paging farm approach is a general-purpose method with no knowledge of the specifics of the application.

The three example problems that we have presented are characterized by good locality of access on each piece of data that gets distributed. In each of our examples, the computation locus changes "slowly" from one piece of data to another. This allows the number of hops to be kept relatively small, which substantially reduces the cost of communication. Adapting our approach to more complex data reference patterns, with code making frequent and less predictable jumps back and forth on widely spread data, will require further research.

References

[1] L.F. Bic, M. Fukuda, and M.B. Dillencourt, "Distributed Computing Using Autonomous Objects", *IEEE Computer*, vol.29, no.8, IEEE Comput. Soc, Aug. 1996. 55-61

[2] William L. Briggs, *A Multigrid Tutorial*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987.

[3] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1991.

[4] George Dramitinos and Evangelos P. Markatos, "Adaptive and Reliable Paging to Remote Main Memory", *Journal of Parallel and Distributed Computing*, **58**, (1999) 357-388

[5] M. Fukuda, L.F. Bic, and M.B. Dillencourt, "Messages versus messengers in distributed programming", *Journal of Parallel and Distributed Computing*, **57**, (1999) 188-211

[6] D. Kotz and R.S. Gray, "Mobile agents and the future of the Internet", *Operating Systems Review*, vol.33, (no.3), ACM, July 1999. 7-13

[7] Thomas J. R. Hughes, *The Finite Element Method, Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, 1987.